For More Awesome Content, you can follow me on youtube -

https://www.youtube.com/@AkhilSharmaTech

Connect with me on ->
Linkedin -> https://www.linkedin.com/in/akhilsails/
Twitter -> https://twitter.com/AkhilAiri

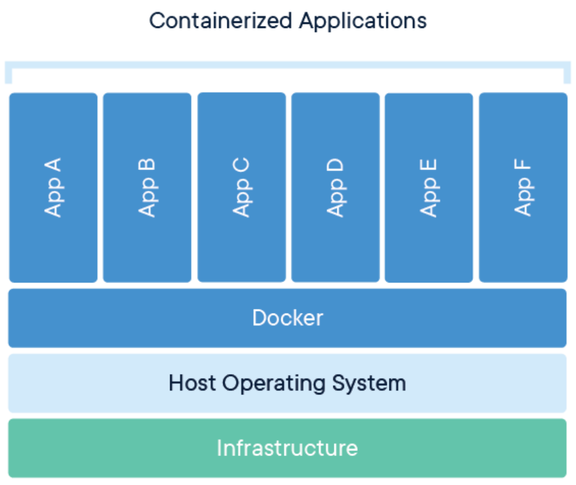
What is Docker?

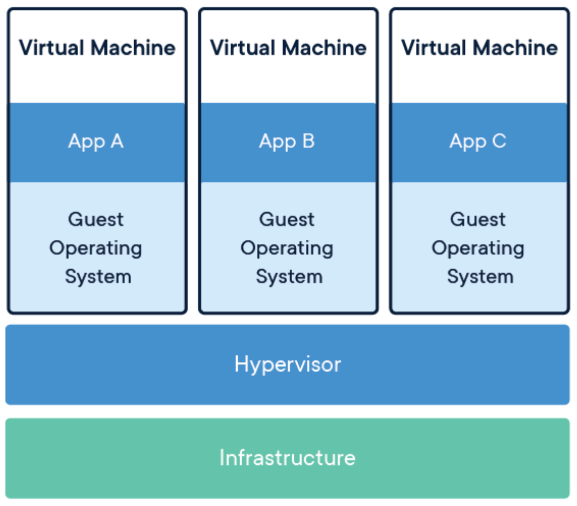
App containerization and isolation.
Maintains integrity of application environment
Saves hundreds of hours of developers time so they can focus only on building application
Shareable environment between developers and between localhost and all other types of servers like QA, dev, prod - the environment will remain same everywhere.

Why to use docker?

One developer gives project to other developer - even after installing all dependencies, project doesn't work properly.
One developer makes project live on server - it works on his machine but not on server - this problem can be solved as well with docker.
Code works fine on QA, dev server but not on prod server - this problem can be solved as well.

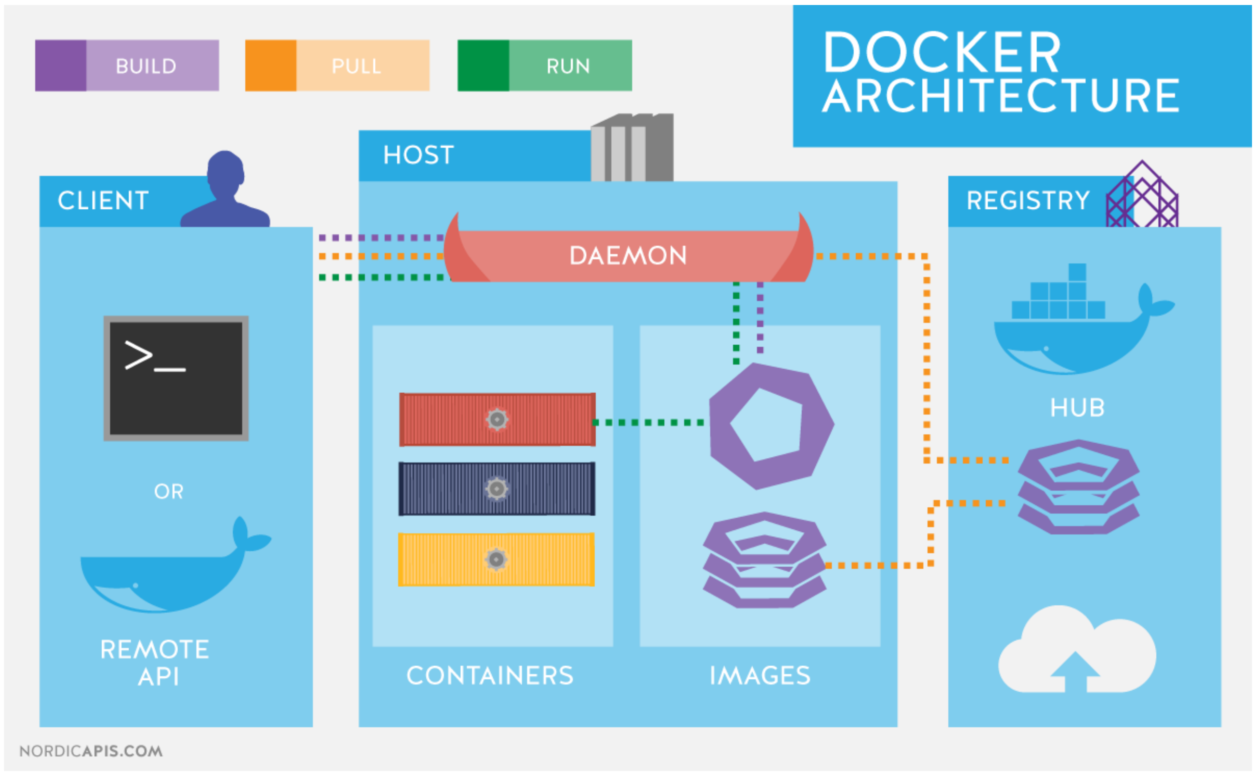
How does docker work?

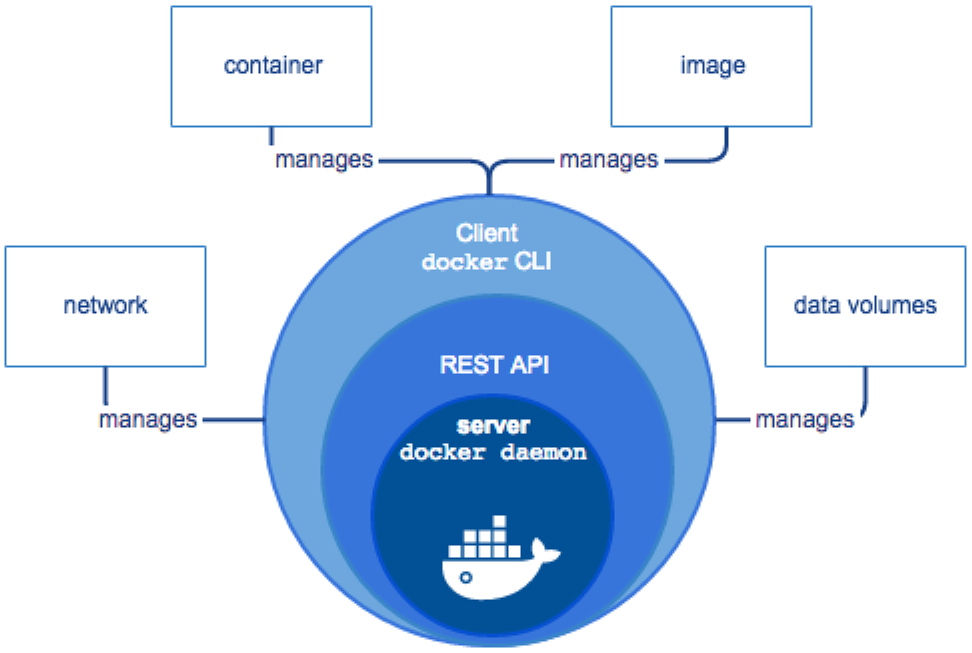




Works on process layer of an OS and not V.M level, this means each container requires much less space, ram and processor to run.

Docker Architecture



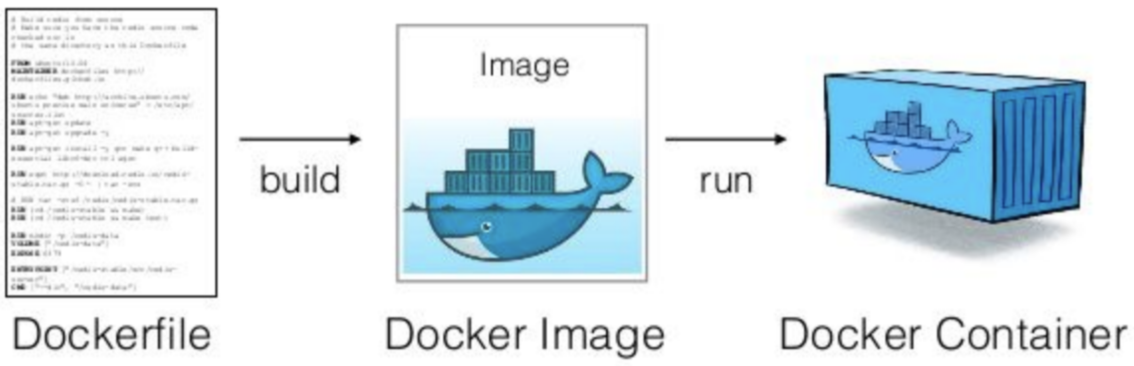


What are images?

Abstraction of a container. This is the file that explains to docker daemon as to what type of container has to be built. Think of this as the plan of a building. We can draw on paper, that the plan of a building will have these many things and from this the building is actually built. The building is the container.

What are containers?

Containers contain applications. Either the entire application can be present in one container or you can have different containers for different applications. So, you can put your nodjs server in one container, mongodb in another and reactjs in different container and then link them together. Or you can just make one big ubuntu container and install the entire project in it and launch that as well. It is usually good to have different containers for different layers because we can increase or decrease particular containers based on traffic and load.
EVERY DOCKER CONTAINER HAS IT'S OWN IP ADDRESS AND PORTS that are exposed.



What is docker hub?

Place which contains all of the images in one place. You can get access to multiple images that you can use or publish your private images for your friends to use. Think of this as github.

Commands

Sudo docker info

Tells you all the information of docker such as how many images, containers etc. are running currently

Sudo docker version

Tells you the docker version installed currently on your machine

Sudo docker --help

To view all commands in one go - shows the different parent modules and the commands associated with them.

Sudo docker image ls (earlier docker image)

Gets list of all images on your local system right now

Sudo docker container ls (earlier docker ps)

Gets list of all containers that are currently running.

Sudo docker container ls -a (earlier docker ps -a)

Shows all containers that are running or have either run in the past, shows all details of them

3

Sudo docker run ubuntu or run nginx

This command is for running a container from an image. Please remember that a container can only be run from an existing image. first this looks for an image in your localhost, if it doesn't find it, it looks for it on docker hub, pulls it from there and then runs the container based on the image.

Any command that starts a docker container, also automatically quickly stops the container.

Sudo docker run ubuntu sleep 30

This command runs the container for 30 seconds, we can go over to another terminal and see whether this container is running or not.

Sudo docker container kill <id>

This will kill or stop a container that's running right now. Don't need to give entire id, just first 2-3 unique numbers is enough.

Sudo docker container rm <id> <id>

This will remove the container with the mentioned id. Can also give 2,3 ids at the same time ro remove multiple containers.

Sudo docker container start <id>

You can get the list of all containers that have existed on your machine by running sudo docker container ls -a and then can start that container. This command takes id, whereas run command takes name of image - this command starts a container, it does not build a container from an image, neither does it automatically pull an image and then create container from it - that's done by run command. This just starts an existing container.

Sudo docker container stop <id>

A container that's started for 30 seconds can be stopped. Difference between kill and stop is that stop is a polite stop and is recommended for stopping containers. If your process is stuck somewhere, then you use kill to forcefully stop a container - just like we forcefully shut down our laptop.

Sudo docker container run -d ubuntu sleep 30

Builds and starts container from an image but detaches it to the background. It does not interrupt your terminal. 90% times we have to use detached background containers only.

Sudo docker container run -it ubuntu /bin/bash

This command takes you inside a container as it is called as the interactive terminal for going inside the particular container. Please remember not to run this command with -d as then it will detach this from the current process and you won't be able to access this, then you have to stop the container and run again. You can install whatever you want in the container from here. You can run ls, apt-get update, apt-get install nodejs etc. any command can be run here.
Type EXIT to exit a container and go back to localhost.

Sudo docker container rm $(docker container ls -aq)

Passes the first command which is list all containers into container rm. Same can be done with container stop command as well.

Sudo docker container inspect <id>

This command works for a container that's running in the background. This helps us to find out important details such as the ip address of a particular container. For example - you can run a container like nginx and then enter the ip address in your laptop's browser to see the response of that container on that ip address.

Sudo docker container logs <id>

Shows the entire logs for a particular container

Sudo docker container top <id>

Shows all the processes running inside a docker container

Sudo docker container stats

To see memory being eaten by all of the containers

Naming a container

Sudo docker container run -d -p 3000:80 --name test_1 nginx

Any request going to our machine's or any virtual machine's (on which the docker container is running) port 3000 will be mapped directly to the docker container's port 80

This can be any port, not necessarily has to be 3000

Example, start nginx container

Docker container exec -it <id> /bin/bash

This command executes /bin/bash inside the container of the id mentioned above. You can execute any command.

Docker container restart <id>

If a container is stopped, you can use start command to start it again. But if it is running, and you just want to restart it, you can use this command instead of using stop and then start

Docker container attach <id>

If a container is running in the background, with this command, it can be brought to the front. It is the opposite of detach.

Docker container wait <id>

This waits for a container to stop and when it actually does stop, this command gives us the exit status of that container. This is a highly effective command for debugging - tells you why the container exited and it's exit status.

Docker container pause <id> and Docker container unpause <id>

Docker container prune -f

All containers get deleted. -f is used so that docker will not ask you for confirmation

Docker container port <id> / Docker container port <name>

Tells us the port to which this is being mapped. By default, even if the container's port is open, it doesn't get mapped to the host machine, unless we specify -P or give a specific -p port. So, this port only shows the host machine's port that is getting mapped to this container.

Docker container create <image_id>

This command just creates a container from an image, doesn't start it. So, we can say that run command, not only creates but also starts and in addition to that, it also pulls image from docker hub, which can be done by pull command. So run is a mix of pull, create and start.

Docker container diff <id>

This command tells us what all files changed inside that container. Helps us compare so that we can know that these all files have changed. This is useful when we merge code and bring it inside docker, now we can find out that this new version or merge has made all these changes. Can be used when another developer commits things to your container.

Watch 'docker container diff <id>'

This command keeps watching for any changes, very helpful command. So let's say a docker container is running a docker file and many steps will get executed so you can keep watching if things are happening as planned from another terminal.

Docker container cp test/ <container id>:/tmp/

This command takes all contents of the test folder that are on your PC and copies them inside the container with the id mentioned at the location that is mentioned.

Docker container exec -itd <id> /bin/bash

It is best to run the command on top with this particular command running in the background (with detached flag(as then you can check whether the file was copied in the container or not.

Docker container export <id> >name.tar

Exports a container in tar format that can be used by another developer. To view the container just created, run the command "ls -lh" and you will be able to see it. Where -l is a long listing flag and -h is a human readable format flag. The other developer first needs to make an image from this tar file and then can use it.

Docker container export <id> -0 name.tar

Same as above command, just different syntax.

Docker image import <exported_container_name.tar> <new_image_name>

This helps us to import the tar file given to us by our friend as an image.

You can run containers from this image now,

Docker container commit --author “author_name” -m “this is test commit” <container_id> <mew_image_name>

This command helps us to create an image from a container, - m flag is to put a commit message. This image can then be shared with friends or uploaded to docker hub.
Please note that till now we were making a tar file and importing it to make an image, this command creates an image directly from the container.

Docker pull <image_name>

To pull an image form docker hub

Docker login

You can login to your dockher hub from the terminal

Docker image tag <image_name> hub.docker.com/akhilsharma/<image_name_for_dockerhub>

Helps to tag the image before pushing.

Docker push <image_name_for_dockerhub>

Push image to docker hub

Docker image ls --format ‘{{.ID}}, {{.Repository}}’

Once the number of images increases, we then need to see in a formatted manner with comma separated. You have to give space after “format”

Docker image history <image_name>

Gives you entire history of the image - when it was created etc.

Docker rm -f <image_name>

Deletes the image either on your laptop or it can also be on docker hub, so for that you can write like this akhilsharma/ubuntu_image

Docker image prune

All unused images in the laptop will be deleted (dangerous command)

Docker image save <image_name> **>** <image_name.tar>

Creates a tar file that can be saved or sent to your friend from an image. ***Please note*** - remember to mention the version number here. Otherwise it will take all ubuntu images and compress them into a tar file and not just a particular ubuntu file

Docker image load **<** <image_name.tar>

Loads the tar file into an image

Docker image ls -a | wc

Gives you the count of images

Docker images -q

Gets only the ids of the images without the other data. This is helpful because many times the number of images increase a lot and thereby info becomes too much.

Difference between docker image save and docker container export

Docker image save command saves all layers and compresses them to a tar file
It is important to note that when you export a container into a tar file, all the volumes that are attached to it - these won't be included in the tar file, only container data will be included.

Docker logs <container_id>

In some containers this doesn’t work, if the image used was too basic and didn’t have json drivers in it. (have tested on mongo containers, worked perfectly).

Flags -

Docker logs --tail 100 <container_id>

Gives the last 100 lines.

Docker logs --since 10 <container_id>

Shows the last 10 minutes logs.

Docker container inspect --format={{.LogPath}} <container_id>

Creates a file from the logs and stores it in your laptop, you can grep this file for errors later.

Sudo grep <the_folder_path_returned by_the previous command>

Will simply search for errors and give you from the log file created.

Docker rmi $(docker images -q --filter “dangling=true”)

Removes all dangling files, if you just put docker images --filter “dangling=true” then it just shows you all the dangling files.

Running Jenkins

Sudo docker pull jenkins

To get the jenkins image

Sudo docker run -p 8080:8080 -p 50000:50000 jenkins

This command, maps the localhost or host machine’s port 8080 to the 8080 port of the container that is running jenkins and also port 50000 of localhost or host machine to port 50000 of jenkins

Sudo docker run -p 8080:8080 -p 50000:50000 -v /users/akhil/desktop/Jenkins_home:var/jenkins_home jenkins

To give a persistent storage space on your localhost or host machine that corresponds to home inside jenkins server - it is similar to port mapping only. Sometimes above command may not work, so this command is more stable. We can also give it a docker volume instead of having it store data inside the docker container.
You can then access jenkins dashboard from localhost:8080

Sudo docker run --name myjenkins1 -p 8080:8080 -p 50000:50000 -v /users/akhil/desktop/Jenkins_home:var/jenkins_home jenkins

Same command as above but we have now given it name also.

Git with jenkins -
https://www.youtube.com/watch?v=Jo5SyGL5QN0&list=PLhQbSuZ3t7xXNvEEN19QSEOEpkAJ3s-G_&index=2&t=0s

https://www.youtube.com/watch?v=bGqS0f4Utn4

Auto deploy with jenkins -
https://www.youtube.com/watch?v=j5D8SLxn6YA

Playlist for build+test+deploy using jenkins -
https://www.youtube.com/watch?v=HZOIl16W0oY&list=PL6tu16kXT9PqIe2b0BGul-cXbmwGt7Ihw

Use jenkins with docker for any type of project such as nodejs -
https://www.youtube.com/watch?v=kKytd5IaE8Q

Docker file topic

Vim needs to be installed.

Then vim Dockerfile

(vim creates a new dockerfile, remember we haven’t put any extension)

EXAMPLE 1 ->

```
FROM ubuntu:16.04
MAINTAINER akhil sharma <akhilsails@gmail.com>
RUN apt-get update
CMD [“echo”, “hello world from my first docker image”]
LABEL name=”akhil sharma”
LABEL email=”akhilsails@gmail.com”
ENV NAME akhil
ENV PASS akhil
WORKDIR /tmp
```

: (this command takes you to the end of the document)

:wq! (this command helps you exit the vim document)

:set number (this command gives number to each line of vim)

Docker build**.** OR Docker build <file_path> OR Docker build -t <image_name>**:**<version> **.**

This will create a docker image from the docker file, docker file is a list of specifications and commands that we can tell docker to execute. Build file can be used by navigating to the folder where the dockerfile exists or specifying the pathname. This command is most stable and predictable when you give -t tag

You can then run this image but remember to give the version tag while running the image, otherwise it is confused for any general image and docker won’t be able to find it.
CMD is the command that runs first as soon as the container is built from the image that’s built using this file.

One interesting observation here is that installation of technologies and configuration based on instructions written in docker file happen at the image building level and not at container running level.

One important thing to note is that if you have an image on your laptop and you build a new image from a dockerfile that doesn’t have any changes in comparison with an existing image, the i.d given to the new image formed will be the same as the existing image - this is a clever feature of docker.

When we write multiple lines in dockerfile and build an image from that, an entire layers of images are built to execute each line one by one. The final result is an image that will contain all the executed commands you have written but the intermediary images created to reach the last one can be seen using this command -> docker image ls -a

Another important thing to note is, that we can write multiple lines in one line by putting && so that they can be executed in single line, it is not best practice to put multiple RUN commands one after the other, also, when we make any changes to the bottom of the dockerfile, only the last line is executed as earlier commands are retrieved from cache but if we make changes to the first line or any middle line, all commands below it will be executed from scratch and not from cache- this happens due to layering which is basically the order in which things need to be installed, for example, apache can be installed only after linux is already installed.
Workdir command changes work directory and ENV, helps us define the credentials for the environment.

This is the complete list of instructions that you can write in a docker file -
https://github.com/wsargent/docker-cheat-sheet#dockerfile

This link is the official link from docker and has details on the docker file -
https://docs.docker.com/engine/reference/builder/#environment-replacement

EXAMPLE 2 ->>

```
FROM ubuntu:16.04
MAINTAINER akhil sharma <akhilsails@gmail.com>
RUN apt-get update
CMD [“echo”, “hello world from my first docker image”]
LABEL name=”akhil sharma”
LABEL email=”akhilsails@gmail.com”
ENV NAME akhil
ENV PASS akhil
WORKDIR /tmp
RUN useradd -rm -d /home/ubuntu -s /bin/bash -g root -G sudo -u 1000 ubuntu USER ubuntu
WORKDIR /home/ubuntu
```

In the above file, we have created a user and added him to the root group.

```
FROM ubuntu:16.04
```

```
MAINTAINER akhil sharma <akhilsails@gmail.com>
RUN apt-get update
CMD [“echo”, “hello world from my first docker image”]
LABEL name=”akhil sharma”
LABEL email=”akhilsails@gmail.com”
ENV NAME akhil
ENV PASS akhil
WORKDIR /tmp
RUN useradd -rm -d /home/ubuntu -s /bin/bash -g root -G sudo -u 1000 ubuntu USER ubuntu
WORKDIR /home/ubuntu

RUN whoami > /tmp/1stwhoami.txt
USER $NAME
RUN whoami > /tmp/2ndwhoami.txt
RUN mkdir -p /tmp/project
COPY testproject /tmp/project/
```

This dockerfile shows how we can change user between two different commands. After building the image from this docker file and running the container, you can cat 1stwhoami and cat the 2ndwhoami file and you will be able to see the changed user.

```
FROM ubuntu:16.04
MAINTAINER akhil sharma <akhilsails@gmail.com>
RUN apt-get update
CMD [“echo”, “hello world from my first docker image”]
LABEL name=”akhil sharma”
LABEL email=”akhilsails@gmail.com”
ENV NAME akhil
ENV PASS akhil
WORKDIR /tmp
RUN mkdir -p /tmp/project
COPY testproject /tmp/project/
```

Create a folder testproject in the directory where you have your dockerfile and fill up this folder with random files.
The mkdir command creates a directory in the tmp folder called project. Inside this folder, we paste all the files from the testproject folder that exists on our own machine.

You can use ADD command in place of COPY too, but ADD account can extract the inner files and copy them, COPY command simply copies the entire file, does not extract files and copies them.

```
FROM ubuntu:16.04
ENV NAME akhil
ENV PASS 123
RUN mkdir -p /var/run/sshd
RUN apt-get update
RUN apt-get install -y openssh-server
RUN useradd -d /home/akhil -g root -G sudo -m -p $(echo “$PASS” | openssl passwd -1 -stdin) $NAME
CMD [“/usr/sbin/sshd”, “-D”]
```

This will start the secure shell in the container and start it in the background. Build this image, and run the container in the background. Now we can ssh into this container, just like we ssh into a virtual machine. So, we have to type -

Ssh akhil@<ip_Address_of_container>

(you can can ip address of container by inspect command, after this command executes, it will ask you for the password, which has been defined as akhil in the docker file)

```
FROM ubuntu:16.04
LABEL name=”akhil sharma”
LABEL email=”akhilsails@gmail.com”
ENV NAME akhil
ENV PASS akhil
RUN useradd -d /home/akhil -g root -G sudo -m -p $(echo “$PASS” | openssl passwd -1 -stdin) $NAME
EXPOSE 22
CMD [“/usr/sbin/sshd”, “-D”]
```

Expose 22 command will expose the port 22 of this container, when you run the container and ‘ls’ it, you will be shown the port of the laptop that maps to the port 22 of docker.

To see the ip of your machine, type **ifconfig**

To check this, type -

Ssh akhil@<ip_address_of_laptop> -p <port_that_was_shown_in_dockerlscontainer_command>

Best Practice -

1. If you have to make a new image from an existing image, then download that image, open the terminal and run the commands and if they work, only then add them as steps

to the new dockerfile and only then keep this new file for the new images. Do not directly write the commands in the dockerfile.

Important points to note -

1. ***When you run an ubuntu container, you can see in the docker file of the ubuntu image that the last command is CMD['bash'] whereas if you run an nginx, mysql etc. container you will see in the dockerfile that the last command is CMD['nginx'] while SQL and NGINX are commands that will be run and hence the container keeps running as these are command prompts of nginx and sql respectively, in case of ubuntu, bash is a listener and hence it listens for commands and if it doesn't find any, it stops. This is why you see the different when you run an ubuntu container as opposed to an nginx container.***
2. ***This is why when we run the ubtuntu container, we need to run the sleep command with it - sleep 30 or 60 so that it can wait for this time for commands and then exit and not automatically and directly exit***
3. ***If you don't want to enter the sleep command again and again when starting the container, then you can simply create a new dockerfile by taking the original ubuntu image like so -***
   FROM ubuntu
   CMD sleep 30
   This will take reference of the ubuntu file and add sleep 30 command to it, then this will become your new dockerfile. This is the beauty of docker, you can take any existing image from the internet and you can view its dockerfile on github and best practices from that, you can also import everything from the image and make any additional changes or customizations to that
4. ***In CMD, you can have a direct command like sleep 30 or you can pass JSON like CMD ["sleep", "30"], it is important to note that the executable needs to be first and the parameter needs to be next, it can't be the other way round and also, both need to be comma separated.***
5. ***Let's say you create an image from this dockerfile and run the container, now if you want the container to run an extra 10 seconds on top of the 5 seconds that are already mentioned by default, you have to again specify <docker container run ubuntu-sleeper sleep 10> this beats the purpose completely of the CMD, this is where ENTRYPOINT comes in, it lets you pass on variables from command line. So, in the docker file, you need to make this change -***

   FROM ubuntu
   ENTRYPOINT ["sleep"]

And in the docker command you can just say <docker container run ubuntu-sleeper 10>, you don't have to say sleep 10, because sleep is already the entry point.

Now, if you don't specify an operand in the container run command, it gives an error, so, if you want to specify a default value, you can use both entry point and CMD

FROM ubuntu
ENTRYPOINT ["sleep"]
CMD["5"]

This will ensure that the default value is 5 seconds for the sleep command.

Docker Volume Concept

We can store data that is required in a docker volume. The volumes are separated from a container and even if containers gets deleted, the volumes still stay. We can assign a volume from a deleted container to a new container as well. This is very handy as for example, you have a ndoejs container and you have a mongodb volume to store the data, now the nodejs needs to be updated, we can delete that container and make a new container with the updated nodejs version and this new container can be connected with the old volume so that there is no change or problem at the data level.
It is important to note that when you run the prune volumes command, only volumes that are not in used, i.e not connected to a container are delete and the volumes that are attached to a container are retained. If a container is running and you try to delete it's volume using rm -f command, it is not possible, also, if a container is stopped and you try to delete it's volume, this is also not possible - you have to kill/stop and delete the container, only then you can delete the volumes associated with it.
There is another type of volume called the bind. What this does is, it binds a folder or storage on your localhost or the host machine of the docker to a storage in the docker container, and whatever updates you do on this folder on your localhost machine are automatically transcended into the folder in the docker machine. This is an important concept and is quite a handy tool. This means, you don't have to keep updating changes by copying new files inside the docker container.

For mongodb -

Docker pull mongo

Docker image inspect mongo

This will show us in which folder does the volume usually get mounted. For example, for this particular image, it is /data/db and /data/configdb

Docker run -d -p 27017:27017 -v ~/test-folder:/data/db --name mongo_test mongo:latest

(we have kept a bind mount on our laptop because even if the container gets deleted, we can use the bind mount to get the data)

Sudo netstat -nltp

Many times ports are already in use and we get an error when we try to expose a particular port, to get info of the same, we use this command

Sudo kill <process_name_for_the _port>

You have to give the process name for the port to kill the process on that port

Docker exec -it mongo_test bash (goes inside this container)

Mongo (this command will open mongo cli)

Show dbs (shows list of db using mongo cli)

Docker container inspect mongo_test

You will be able to see the mount - source (from laptop) and destination (on the container) for this container. You will also see the folder in which you have to mount the volume.

An important point -

If you create a container like mysql or mongodb, they automatically create their own volumes and you can inspect the container to see the folder in which a volume will usually be mounted.

You can inspect an image of mongodb or mysql to determine the path where they attach the volumes.

You can also inspect the volume,
In the mountpoint you will also see the folder on the host directory in which the volume on the host machine lives. For example the folder mentioned is var/lib/docker/volume/abcd
In the host machine, you can go to that folder and view the data of the volume.

Sudo docker volume hello

To create a new volume, hello

Sudo docker volume rm hello

To remove the volume

Sudo docker container run -itd -v <volume_name>:/var/lib/mysql mysql

You can create your own volume and attach it, and here, <volume_name> is the volume you created.

Where mysql is the name of the container and /var/lib/mysql is the default place where the volume is attached for this container, you can view that by image inspect and voume_name is the volume name you want to attach to it - if you don't do this, a default volume is created and attached, the command above can be used in instances where you want to attach a volume from a previous container.

Docker Networking concept

There are many types of networks available to us - bridge, overlay, macvlan, null, host

When we use host networking - we are telling the container to completely reciprocate the host's network, same ports etc. will be used but only difference is that you will be getting a seggregation due to the presence of the container.

If you just run a container, by default, it will run on the bridge network. This is the default network that comes out of the box. The bridge network exists inside a particular host, it is the network that all the containers inside one host use to communicate with one another and this is completely isolated from the outside world or the outside docker engines/host cannot access it. This is why we use the -p 80:80 port mapping command, to tell docker to map port 80 of the host to port 80 of the container so that these can communicate outside the host container as well. You don't have to expose identical ports, you can expose port 5000 of container to port 80 of the host.
To make things secure, ensure you are not unnecessarily opening up ports.

Bridge and host network are single touch point networks only - this means that in host network, only one container can take the identical mapping of the host and in bridge network, while the containers can talk to each other, only one of them's port can be mapped to the outside world, the problem comes in when we have multiple containers inside a host that need to talk amongst themselves but also to the outside world. This is where overlay comes in.
The overlay network is encrypted and keys are rotated every 12 hours between the containers.

Null network allows you to route your address coming into the container to a blackhole. This is only used by iot engineers.

Macvlan network provides us a physical IP/TCP stack so that containers think that an actual computer is talking to them. This is used by old systems that only understand talking to physical servers.

A good design pattern when having docker swarm and overlay is to have two different managers - one for control and one for data - this means, the manager you talk to to tell what you want the cluster to do should be separate and the manager that then communicates with all of the containers to pass on the message and tell them what to do and manage their tasks should be separate.

UI management for docker swarm - https://swarmpit.io/ - kubernetes already comes with its GUI, so better to use kubernetes

Docker network ls

To see the list of networks - you will see the default networks - bridge, null, host etc.

Docker network inspect bridge

To see which all containers are attached to the bridge network

If you start any container, it will automatically get attached to the bridge network and you can determine that by running the above command again.

Docker container run -itd -P nignx

The P command opens and assigns a random port to the container from the host machine

After this if you run the docker container ls command, you will see the port that has been assigned to this container and you can then check this by going to the browser on localhost ip address and then the port assigned and you will be able to see nginx running (which is actually the container). In earlier examples, we have seen how to assign a port manually to a container by doing 80:80, -P is just for random assigning.

Docker network create -d bridge test

You will be creating a new bridge network here, other than the one that already exists.You can view the list by docker network ls and you will see this new bridge network with the name test.

Docker container run -it --network test ubuntu /bin/bash

Creates a container and connects it to the bridge network called test that we created in the command above.

If you were to simply run the **docker container run -it ubuntu /bin/bash**, then you attach it to the default bridge network running.

To see which container is attached to which network, run the ifconfig command on your machine to see veth values. Where veth is the virtual network created when a container is running and it is deleted as soon as the container ends.

Once you have created 3 containers - 2 of them connected to this new test network you have created and 1 of them connected to the default network and if you try to ping any of the two test network containers from the default network container - the result will fail - this is because bridge is for internal use only.

If you want to setup networking between these containers of different networks, then you can either assign a port manually or use the -P command to assign random port and then the container from the other network can ping on this or “wget” this - you can apt-get install wget, if you don’t have it installed in the container. You will have to ping the host machine’s ip address at the assigned port from the container on the other network and you will then be able to communicate with this container that has been assigned a port on the host machine.

Docker network create -d host test

It will not allow you to create another network with host as only one can run at any point of time. This command tries to create a host network with the name of test.

Docker container run -it --network=host ubuntu /bin/bash

This command will run a container from the ubuntu image and map it to host network. When you type ifconfig command in this container, the result will be exactly the same as the ifconfig command that you run in the localhost machine. There is no requirement of port mapping here, you will be able to access the container directly from host machine's ip address.

In the locahost machine, if you type

Docker network ls

You will see a host network running, and if you try to create a new host network using the below command -

Docker container run -it --network=none ubuntu /bin/bash

Creates a container which is attached to null network or to no network at all. Name of null network is actually none.

Very important concept -

When two containers are connected to the same network, dns is already enabled between them, which means if you ping the other container from this container, by just giving the ID or the name of the other container, you can start receiving the data, not necessarily you have to go to the ip address of the other container.
In many cases, ping is not installed in the container by default, so you can install it by -

Apt-get install -y iputils-ping

It is important to note that the network bridge that comes as default in docker, in that dns is not enabled but any bridge network that you create by yourself, in that, dns is activated.

You can remove network by a standard rm command and also prune command to delete all networks not in use.
You cannot delete a network to which a container is attached.

If there’s a container attached to the default bridge network for example, you can connect it to any other network, for example a bridge network called test that you have created, by -

Docker network connect test <container_name>

You can disconnect it from the network by using the same command and replacing connect with disconnect.

Docker Registry Concept

By default all our images get stored on docker hub if we push them. But what if we want to store our images on a container that we create in our localhost or host of any virtual machine and use that to upload and download images? This is useful when we have to share images between developers in a secure manner.

Docker production level (important to watch and understand these 4 videos) -

https://www.youtube.com/watch?v=PpyPa92r44s

https://www.youtube.com/watch?v=6jT83IT6TU8

https://www.youtube.com/watch?v=IgQv5chQHyc

https://www.youtube.com/watch?v=AdMqCUhvRz8

Docker Compose Concept

In any folder, create a docker-compose.yml file and type the following -

```
Version: ‘1’
Services:
        Webapp1:
                Image: nginx
                Ports:
                  - “8000:80”
```

Docker-compose up -d

Will build containers based on the specified docker compose file.

Docker-compose down

Will delete /remove all the containers, networks volumes etc. built using the up command.

```
Version: ‘2’’
Services:
        Webapp1:
                Image: nginx
                Ports:
                  - “8000:80”
       Webapp2:
                Image: nginx
                Ports:
                  - “8001:80”
```

This example shows you can create multiple services at the web app layer.

Docker-compose -f docker-compose2.yml up -d

Will startup the file called docker-compose.yml and not the standard docker-compose.yml file. This is done whenever you have more than one compose files.

Docker-compose -f docker-compose2.yml down

To take down all the containers, networks and volumes that were created by this compose file.

Docker-compose create

This command just creates the containers, doesn’t run them. Just like docker container start. So up is the equivalent of run in container terms.

Docker-compose rm

To remove created container (to be used as opposite of compose create command.)

Docker-compose start

To start containers that had been created

Docker-compose stop

Stops the started containers.

Create, start, stop and rm - they don’t do any tasks related to the network, only related to the containers. For network tasks, only up and down will work.

Docker-compose ps

Shows only the containers live for docker compose and not all the other running containers. So this is quite handy.

Docker-compose pause and docker-compose upause

Docker-compose port webapp1

Shows which port the service webapp1 defined in the docker compose file is mapped to

Docker-compose logs   (can also use -f flag at the end)

This shows us all the logs and -f flag allows us to follow the containers.

Docker-compose exec webapp1 ls

Runs the ls command in the container called webapp1 defined in the docker compose file

Docker-compose run webapp1 ls

While exec runs the command mentioned in a running container, run command creates a completely new container and runs the command there.

Docker-compose pull

To pull the images specified from docker hub.

Docker-compose scale webapp1=2 webapp2=4

This command will scale the webapps based on the containers you have specified. Must ensure that you haven't mentioned ports in the docker-compose file. Else conflict will occur because subsequent forming containers will demand the same port as the previous one.

```
Version: '3''
Services:
        Webapp1:
                Build: .
                Ports:
                  - "8000:80"
        Redis:
                Image: "redis:alpine"
```

The build command in the compose file will first build a container from the image that was build from the dockerfile. So, for each project, you will have a dockerfile and a docker compose file. Also, if you notice, in the webapp1, we haven't mentioned any image, but we have mentioned for redis, this is because we want webapp1 to use an image built from dockerfile whereas redis is used as a seaparate database service level and will use the redis:alpine image,

```
Version: '4''
Services:
        Webapp1:
                Build: .
                Ports:
                  - "8000:80"
                Image: 'akhil/node:latest'
        Redis:
                Image: "redis:alpine"
```

If you want to give a name to the image built, you can do that as well.

Below is an example of arguments

```
Version: ‘5’’
Services:
        Webapp1:
                Build: .
                Ports:
                  - “8000:80”
                Args:
                 -NODE_VERSION:4.0
                Image: ‘akhil/node:latest’
        Redis:
                Image: “redis:alpine”
```

Example of the dockerfile below where the argument will be substituted.

```
ARG NODE_VERSION
FROM Node:NODE_VERSION
```

```
Version: ‘6’’
Services:
        Webapp1:
                Build: .
                Ports:
                  - “8000:80”
                Args:
                 -NODE_VERSION:4.0
                Image: ‘akhil/node:latest’
                Networks: appnetwork
                Environment:
                        Name:akhil
                        address:pune

        Redis:
                Image: “redis:alpine”
                Volumes: myredisdata:/ data
                Networks: appnetwork

        Redis2:
                Image: “redis:alpine”
                Volumes: myredisdata2:/ data
                Networks: appnetwork2
        Networks:
                Network1:
                Network2:
        Volumes:
                Myredisdata:
                Myredisdata2:
```

We have also seen how to use enviroment variables in our docker compose file. Now we will see how to use an external env.txt file

```
Env_file:
    -   .env.txt
```

You can also specify everything in a .env file and dockercompose automatically searches for this file.

Docker Swarm Topic

Better to have vagrant instances of all machines so you don't have to install basic things in ubuntu such as docker etc.

In a swarm, there are multiple nodes and they can be either managers or slaves existing in different machines. There can be multiple managers as well, having a lot of managers doesn't necessarily mean that your application will becomes faster, the managers are restricted to this formula (n-1)/2, so if there are n nodes then there will be (n-1)/2 managers atleast because they work on raft distributed consensus algorithm and it is required by this algorithm to have these many managers so that even if some managers go down, there will be someone to take it's place.
Managers can act as workers and workers can act as managers all at the same time. Also, nodes can be demoted and promoted at will. Docker managers should be in this series - 1,3,5,6 etc.

Docker node ls
To view all the nodes in a swarm currently